RIACS

Research Institute for Advanced Computer Science
NASA Ames Research Center

# Modula-2*: An Extension of Modula-2 for Highly Parallel Programs

Walter F. Tichy* and Christian G. Herter

RIACS at NASA Ames Research Center†

RIACS Technical Report 89.34

September 1989

# Modula-2*: An Extension of Modula-2 for Highly Parallel Programs

Walter F. Tichy* and Christian G. Herter

RIACS at NASA Ames Research Center†
RIACS Technical Report 89.34

September 1989

# Modula-2*: An Extension of Modula-2 for Highly Parallel Programs

Walter F. Tichy* and Christian G. Herter

RIACS at NASA Ames Research Center†
RIACS Technical Report 89.34

September 1989

**Abstract**

Highly parallel computers with tens of thousands of processors will be of rapidly growing importance for highspeed computation. Parallel programs for these machines should be machine-independent, i.e., independent of properties that are likely to differ from one parallel computer to the next. In particular, parallel programs should be independent of:

1. memory organization and communication network,
2. number of physical processors available,
3. control mode of the parallel computer (SIMD, MIMD, or MSIMD).

This paper describes extensions of Modula-2 for writing highly parallel, portable programs meeting these requirements. The extensions are:

- Synchronous and asynchronous forms of a **forall** statement;
- Control of the allocation of data to processors.

Sample programs written with the extensions demonstrate the clarity of parallel programs when machine-dependent details are omitted. The principles of efficiently implementing the extensions on SIMD, MIMD, and MSIMD machines are discussed. The extensions are small enough to be integrated easily into other imperative languages.

*Supported by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

†Authors' permanent address: University of Karlsruhe, D-7500 Karlsruhe, FRG.

# 1 Introduction

Highly parallel machines with thousands and tens of thousands of processors are now being manufactured and used commercially. These machines will be of rapidly growing importance for highspeed computation. They also indicate that a fundamental paradigm shift from the sequential to the parallel computer is in progress. This shift is fundamental because it affects virtually all areas of computer science, computer engineering, and computer applications.

Ease of programming will be of overwhelming importance for the acceptance of highly parallel machines. At present, writing highly parallel programs is still a poorly understood and extremely complicated craft. What makes highly parallel programs difficult to write and maintain is that they must deal with a plethora of machine-dependent details such as the memory organization and interconnection network, the number of processors available, and whether the target machine runs in SIMD or MIMD mode. To make parallel programs easier to write, maintain, and port, parallel programming languages must abstract from machine-dependent details and allow programs to be formulated in a problem-oriented way.

The current programming style for machines such as the 65,000-processor Connection Machine[3,7] is best characterized as "interconnection programming". This style involves exploiting details about the interconnections among processors and memory units for squeezing the last bit of performance out of the available hardware. Interconnection programming has the same undesirable properties as assembly programming: Programs are difficult to understand and maintain, and have to be rewritten for every new machine type. The rational for interconnection programming is that the communication networks of today's parallel computers are critical bottlenecks, much too slow compared to the speed of the processors. However, given the youth of the field, parallel memory organization and compiler technology are likely to improve significantly, and might render interconnection programming as obsolete as assembly programming. It therefore seems appropriate to design programming languages in which details about the memory organization and interconnection network are irrelevant. Instead, programs should simply exchange data by reading and writing memory in parallel, while fast interconnection hardware and compiler technology implement efficient data transport. The goal is to let the problem dictate the data exchanges, and not a particular computer architecture.

A related machine dependence involves the number of physical proces-

sors. On most parallel machines today, programmers are repeatedly faced with the problem of simulating a large number of parallel threads of control on a comparatively small number of real processors. The resulting programs are extremely difficult to understand, because the code for multiplexing the processors and for packaging and shifting the data accordingly may obscure even simple algorithms. Instead, the problem and the algorithm should dictate the number of processes to be used, and the underlying runtime system should organize the allocation of data and processes to real memories and processors.

A third issue when programming parallel machines is whether they execute in the modes MIMD, SIMD, or MSIMD. MIMD stands for *multiple instruction streams, multiple data streams* and means that each hardware processor has its own instruction pointer, executing its own program on its own data. Processors run independently of each other, except when synchronizing or exchanging data. SIMD stands for *single instruction stream, multiple data stream* and means that all processors execute the same instructions in synchrony on their own data, or idle for some instructions. An SIMD machine consists of a single control processor and a large number of processing elements. The control processor stores the program and issues the instructions to the processing elements. Because of the synchronous execution and the elimination of many race conditions, an SIMD machine is easier to program than an MIMD machine. An SIMD machine also costs less to build than an MIMD machine. First, it needs less memory, because the program is stored only once. Second, the processing elements are simple arithmetic and logic units without program counters, and therefore cheaper to build than full-fledged, general-purpose CPUs. These savings are important for machines that incorporate tens of thousands of processing elements. The drawback is that an SIMD machine may be difficult to utilize fully: Whenever an instruction is issued, only a portion of the processing elements may actually be in a state where they can execute it; the rest of them idle.

A compromise between MIMD and SIMD is MSIMD, short for *multiple SIMD*. An MSIMD machine is similar to an SIMD machine, except that the single controller is replaced by several, each of which may issue a different stream of instructions. The processing elements can choose dynamically which instruction stream to follow. The underlying assumption is that, although a parallel program may branch out into several independent threads of control, the number of such threads is much smaller than the number of processing elements. For instance, two branches of an IF-statement could

be executed simultaneously on an MSIMD machine with two controllers, while an SIMD machine would first idle one set of the processing elements, then the other. MSIMD may also be viewed as VLIW SIMD, or Very-Large-Instruction-Word SIMD.

It is evident that writing a program explicitly for an MIMD, SIMD, or MSIMD machine is another source of machine-dependence. For example, a program written for an MIMD computer such as the N-Cube will normally not run on an SIMD computer such as the Connection Machine, and vice versa. To preserve portability, parallel programs should be written in such a way that the synchronous or asynchronous parallelism is determined by the problem at hand, not dictated by a particular machine architecture. It is the task of the compiler to map synchronous or asynchronous parallelism to the capabilities of the available hardware. The synchronous and asynchronous language constructs presented below can be executed efficiently on MIMD, SIMD, and MSIMD architectures.

The rest of the paper discusses Modula-2*, an extension of Modula-2[8] for writing highly parallel programs. The extensions abstract from the memory organization and the number of physical processors, and let the programmer choose explicitly between synchronous and asynchronous execution. The necessary extensions were surprisingly small. We chose Modula-2 as a base, because we wanted to start experimenting with a simple language. Similar extensions can be integrated into other imperative programming languages, such as C++ or Ada. We also discuss the principles of how to implement the constructs on MIMD, SIMD, and MSIMD machines.

## 2 Parallel Programming Constructs

The extensions of Modula-2 consist of synchronous and asynchronous versions of a **forall** statement, plus a simple, optional declaration for mapping array data onto processors. Furthermore, the restrictions in Modula-2 on compile-time evaluation of array bounds had to be lifted, to allow for flexible, parallel array processing.

For presenting the syntax of the extensions, we use the EBNF notation of the Modula-2 language definition[8], with keywords in upper case, | denoting alternation, ( ... ) grouping, and [ ... ] optionality of the enclosed sentential form.

## 2.1 Overview of the forall statement

The **forall** statement creates a set of processes that execute in parallel. In the asynchronous form, the individual processes operate independently; the **forall** simply terminates when the last of the created processes terminates. In the synchronous form, the processes created by the **forall** operate in unison, but may branch out into mutually independent subsets and then rejoin. Although the asynchronous form is the more general one, the synchronous form is easier to understand because it causes fewer race conditions, just as a clocked hardware circuit causes fewer race conditions than an unclocked one. Where necessary, explicit synchronization of asynchronous processes is possible with semaphores and the procedures SEND and WAIT, as specified (though not as implemented) in Chapter 30 of [8].

The syntax of the **forall** is as follows.

ForallStatement = FORALL ident ":" SimpleType IN (PARALLEL | SYNC)
StatementSequence
END

The identifier introduced by the **forall** statement is local to the statement and follows the usual scope rules. "SimpleType" is an enumeration or a subrange of another enumeration. The basic enumerations INTEGER, CARDINAL, LONGINT, CHAR, and BOOLEAN as well as any user-defined enumeration may be used.

## 2.2 The asynchronous forall

The action of the asynchronous **forall** statement

```
FORALL C : T IN PARALLEL SS END
```

is as follows.

1. Assume the number of values of type $T$ is $N$. The statement creates $N$ processes, each supplied with a constant $C$ bound to a unique value of $T$.

2. The $N$ processes execute the statement sequence $SS$ concurrently. No assumptions about the relative speeds of the processes may be made, unless explicitly synchronized. The statements in $SS$ may refer to $C$ or any other identifier global to the statement. If several processes write the same global variable, then it is indeterminate which value is eventually stored in it.

3. The **forall** statement terminates when the last of the $N$ created processes terminates.

In the following simple example, an asynchronous **forall** statement implements a vector addition.

```
FORALL i: [0..N-1] IN PARALLEL z[i] := x[i] + y[i] END
```

Since no two processes created by the **forall** access the same variable, no temporal ordering of the processes is necessary. The $N$ processes may execute at whatever speed. The **forall** terminates when all processes created by it have terminated.

Our asynchronous **forall** is a simplification of the **forall** statement found in the dataflow languages VAL and SISAL[6,5]. It can express the same degree of explicit parallelism as its dataflow variants. However, dataflow machines can also exploit implicit parallelism, by detecting at runtime that certain subexpressions are independent, and then executing these subexpressions in parallel. A parallel machine constructed out of numerous, individual dataflow processors might be able to exploit this type of implicit parallelism.

## 2.3 The synchronous forall

The synchronous **forall** statement

```
FORALL C : T IN SYNC SS END
```

differs from the asynchronous form only in that the created processes execute the statement sequence $SS$ synchronously. Roughly stated, synchronous execution means that all processes that follow the same path through the control flow graph execute instructions in lock step. However, processes on differing control flow paths may execute asynchronously. This scheme is not SIMD, since control flow may diverge in conditional statements. For example, consider the synchronous execution of an **if** statement with two arms. First, all processes evaluate the condition synchronously. The evaluation splits the set of processes into two subsets, depending on the result of the condition evaluation. The subset with processes containing the value TRUE then executes one arm synchronously, the other subset the other arm. Both sets may operate simultaneously. Though processes in the same subset operate in lock step, the speed of processes in different arms are incomparable. When both subsets terminate, they are joined again into one set.

The synchronous **forall** statement is a generalization of the **forall** for SIMD machines described by Hillis and Steele[4]. MSIMD machines can execute our synchronous **forall** directly. SIMD machines can also implement it efficiently, because there is no order implied among diverging control flow branches. The lack of ordering permits a process scheduling that greatly reduces the idling of processors. (See Section 3.2 for more details.)

Below is the precise definition of synchronous execution. The definition is recursive and given for each statement type.

**Sequence:** A statement sequence of the form

```
T1;T2; ... Tk
```

is executed synchronously by executing the statements $Ti$ synchronously in sequence.

**Assignment:** An assignment statement of the form

```
L := R
```

is executed synchronously by $N$ processes as follows. First, all $N$ processes evaluate the designator $L$ synchronously, yielding $N$ (not necessarily different) results each designating a variable. Second, all $N$ processes evaluate the expression $R$ synchronously, yielding $N$ (not necessarily different) values. Third, all processes store their values computed in the second step into their respective variables computed in the first step. If the third step results in several values being stored into the same variable, then it is indeterminate which of those values will actually be stored after the assignment terminates.

**if:** An **if** statement of the form

```
IF    E1 THEN TT1
ELSIF E2 THEN TT2
...
ELSE TTk
END
```

is executed synchronously by $N$ processes as follows. First, all $N$ processes evaluate expression $E1$ synchronously. Those processes for

which *E1* evaluates to TRUE then execute *TT1* synchronously, while the other processes execute *E2* synchronously. Those whose evaluation of *E2* yields TRUE then execute *TT2* synchronously, and so on. Thus, each IF and ELSIF clause divides the set of remaining processes into two independent subsets. The processes remaining after the last ELSIF clause (if any) finally execute *TTk* synchronously. No assumptions may be made about the relative speeds of pairs if processes executing different expressions *Ei* or statement sequences *TTi*. The synchronous execution of the **if** statement terminates when the last non-empty subset of processes terminates.

**while:** A **while** statement of the form

```
WHILE E DO TT END
```

is executed synchronously by $N$ processes as follows. Assume processes may be designated either as active or inactive.

1. Designate all $N$ processes as active.
2. All active processes execute expression $E$ synchronously. Those processes, whose evaluation of $E$ yields false, are designated as inactive.
3. If the set of active processes is empty, then the synchronous execution of the **while** statement terminates.
4. Otherwise, the active processes execute statement sequence $TT$ synchronously.
5. Continue with step 2.

**forall:** A **forall** statement of the form

```
FORALL D : U  ... TT END
```

is executed synchronously by $N$ processes as follows.[1] First, all $N$ processes compute the range $U$ in synchrony. Then each of the $N$ processes spawns a new set of processes given by $U$. If the **forall** specifies synchronous execution, all processes thus created execute the

[1] This is a synchronous or asynchronous forall nested within another, synchronous forall.

statement sequence $TT$ synchronously; otherwise, they execute asynchronously. Synchronous execution of the **forall** terminates, when all created processes have terminated.

**WAIT and SEND:** Synchronous execution of a WAIT by $N$ processes causes all $N$ processes to block if any of them blocks. If the $N$ processes have been blocked, they will continue only after all individual processes that caused the blocking have been unblocked by a SEND from other processes. Clearly, the $N$ processes cannot unblock themselves.

The synchronous execution of expressions, designators, procedure calls, **case** statements, **repeat** statements, **for** statements, **loop** statements, **with** statements, **return** statements, and **exit** statements is defined analogously. Of special importance are procedure and function calls, because they allow multiple, synchronous subprogram invocations. The definitions are omitted here for the sake of brevity.

## 2.4 Example

Consider the problem of summing the elements of a vector in parallel. By using a recursive doubling technique, the sum can be computed in $O(\log N)$ time, where $N$ is the length of the vector. Figure 1 illustrates the process.

The recursive doubling technique operates basically by computing partial sums of length $2, 4, 8, \ldots N$. There is a one-to-one mapping between process numbers and elements of the vector. By inspecting the assignment statement we note that only process $i$ will update the $i$'th element of the vector. In the first iteration, all odd numbered processes are disabled by the **if** statement, since that statement has no second arm. Thus, only the even numbered processes update their respective vector elements. Each of those processes does so by retrieving the element to the right of $V[i]$ and adding it to $V[i]$. (The second condition in the **if** statement makes sure that the last process does not attempt to access a non-existing vector element.) In the next iteration, only the processes divisible by 4 will update their values, but this time they reach for elements that are a distance of 2 away. These are the even numbered elements. Note that these elements already contain sums of subvectors of length 2. The result is that now the updated array elements contain partial sums of length 4. This process continues by doubling the length of the partial sums in each step, until $V[0]$ contains the desired result.

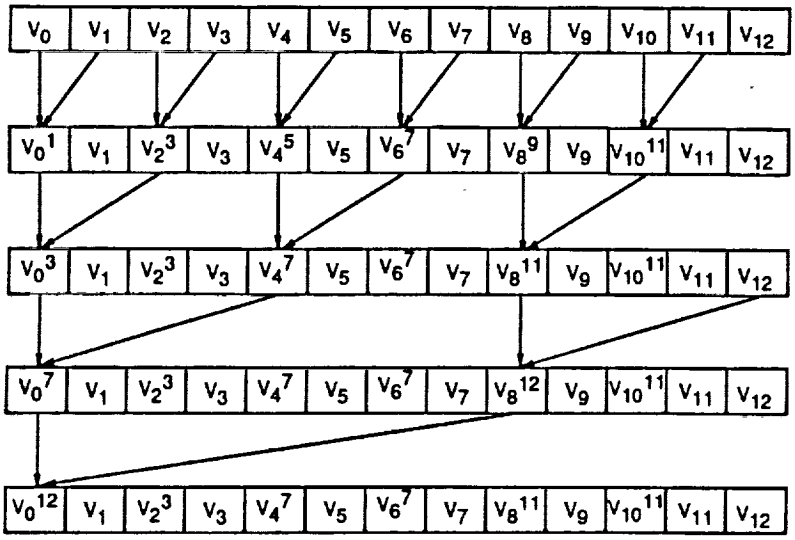


Figure 1: Computing the Sum of a Vector

```
VAR V :       ARRAY[0 .. N] OF REAL;
VAR stride:   CARDINAL;
BEGIN
   stride := 1;
   WHILE stride <= N DO
      FORALL i : [0 .. N] IN PARALLEL
         IF ((i MOD (stride*2))=0) AND ((i+stride)<=N) THEN
            V[i] := V[i] + V[i+stride]
         END
      END;
      stride := stride * 2
   END (* sum in V[0] *)
END
```

Note that the process selection is such that in each iteration, none of the processes interfere. Each process reads and writes its own pair of vector elements. Thus, we can use the asynchronous form of the **forall** statement. The only requirement is that all processes complete before the next iteration commences, but that property is assured by the semantics of the **forall**.

For illustrating the use of the synchronous **forall**, consider interchanging the **while** and **forall** statements in the above program. How would that change the execution of the program? First, each process would now control its own loop, so the loop control variable *stride* must be replaced by an array. Furthermore, the individual processes must now be constrained to execute synchronously. Otherwise, we would obtain unpredictable results, because the processes may overtake each other arbitrarily. For instance, one process might read a vector element that has not yet been updated, or it might overwrite a vector element whose old value is still needed. The synchronous **forall** guarantees that no such interference can happen. The resulting program is below.

```
VAR V :      ARRAY[0 .. N] OF REAL;
VAR stride: ARRAY[0 .. N] OF  CARDINAL;
BEGIN
   FORALL i : [0 .. N] IN SYNC
      stride[i] := 1;
      WHILE stride[i] <= N DO
         IF ((i MOD (stride[i]*2))=0) AND ((i+stride[i])<=N) THEN
            V[i] := V[i] + V[i+stride[i]]
         END
      END;
      stride[i] := stride[i] * 2
   END (* sum in V[0] *)
END
```

This program could be transformed again in such a way that not all processes execute the loop for the same number of steps. Merging the condition of the **if** statement into the condition of the **while** statement would stop each loop at the right time. Yet another transformation would use $N$ semaphores to control the summing process asynchronously.

## 2.5 Allocation of array data

Co-location of data with the processors that operate upon them is important for parallel machines without uniform access time to memory locations. Poor alignment of data and processors may cause excessive communication overhead. We therefore provide a simple construct for controlling the allocation of array data to the available processors. This construct is optional, and does not change the meaning of a program; it affects only performance.

A compiler for a machine with uniform memory access time may ignore the construct.

The allocation of array data to processors is controlled with one allocator per dimension. The modified declaration syntax for arrays is as follows:

ArrayType = ARRAY SimpleType [allocator]
{"," SimpleType [allocator]} OF type

allocator = LOCAL | SPREAD | SCATTER

If the allocator is missing, it is assumed to be SPREAD. The interpretation of the allocator is as follows.

**LOCAL:** Allocate all elements of a dimension marked LOCAL to a single processor.

**SPREAD:** Distribute the elements of a dimension marked SPREAD over all available processors. Elements whose indices differ only by unity in the marked dimension must be allocated to the same processor, as far as that is possible given that all available processors should be used.

**SCATTER:** Distribute the elements of a dimension marked SCATTER over the available processors. Elements whose indices differ only by unity in the marked dimension must be allocated to different processors.

As an example, consider the following declarations.

```
A: ARRAY [1..L] SPREAD  [1..M] LOCAL OF T
B: ARRAY [1..L] SCATTER [1..M] LOCAL OF T
```

The LOCAL allocator forces each row of $A$ and $B$ into one processor. If the number of available processors, $P$, is larger than $L$, then each row is allocated to exactly one processor in both cases. If $1 < P < L$, then row $r$ of $A$ is allocated to processor $((r-1) \div \lceil L/P \rceil)$, while row $r$ of $B$ is allocated to processor $((r-1) \bmod P)$. Thus, SPREAD assigns sequences of $\lceil L/P \rceil$ successive rows to a single processor, while SCATTER distributes these sequences over the $P$ processors.

For a multidimensional array, there is at most one dimension where the difference between SPREAD and SCATTER matters. Consider a multidimensional array $C$ with $n$ dimensions.

```
C: ARRAY [ln..un] allocn ... [l1..u1] alloc1 OF T
```

$$di = \begin{cases} 1 & \text{if } alloci = LOCAL \\ ui - li + 1 & \text{otherwise} \end{cases}$$

$$D(k,l) = \begin{cases} \prod_{i=k}^{l} di & \text{if } k \leq l \\ 1 & \text{otherwise} \end{cases}$$

Determine the largest $m (1 \leq m \leq n)$ such that

$$P \leq D(m,n)$$

where $P$ is the number of available processors. If no such $m$ exists (i.e., $P > D(1,n)$), then there are enough processors to distribute all elements of dimensions marked SPREAD or SCATTER to different processors. (There is no difference between SPREAD and SCATTER in this case.) If $m$ exists, it identifies the dimension where the difference between SPREAD and SCATTER applies. If the allocator of that dimension is SPREAD, then the array elements $C[j_n,\ldots,j_m,\ldots]$ and $C[j_n,\ldots,(j_m \pm 1),\ldots]$ must be allocated to the same processor, as far as that is possible, given that all available processors should be used. If the allocator is SCATTER, then any two such array elements must be allocated to different processors.[2]

Dimensions higher than $m$ that are marked SPREAD or SCATTER are simply distributed over the available processors. Dimensions lower than $m$ are automatically treated as LOCAL, since there are no additional processors available to distribute the data. An implementation may also map the dimensions $m$ and lower into one dimension (i.e., "unroll" them into one, long vector) and then treat the new dimension according to the allocator of dimension $m$.

The function $F$ defined below provides a suitable mapping of elements $C[j_n,\ldots,j_1]$ to processors numbered $0 \ldots P-1$. Many other choices are possible, depending on the interconnections among the processors.

$$F(m,j_n,\ldots,j_1) = \begin{cases} G(m,j_n,\ldots,j_1) \bmod P \\ \qquad \text{if } allocm = SCATTER \\ G(m,j_n,\ldots,j_1) \div \lceil D(m,n)/P \rceil \\ \qquad \text{if } allocm = SPREAD \end{cases}$$

[2] If $P = D(m)$, there is again no difference between SPREAD and SCATTER.

$$G(m, j_n, \ldots, j_1) = \sum_{i=m}^{n} D(m, i-1) \times S(i) \times (j_i - u_i)$$

$$S(i) = \begin{cases} 0 & \text{if } alloci = LOCAL \\ 1 & \text{otherwise} \end{cases}$$

Function $F$ can even be used in the case where $P > D(1,n)$, by setting $m = 1$.

The SPREAD allocator is used to minimize communication overhead in case of nearest-neighbor communications, while still utilizing all available processors. The SCATTER allocator can keep processor utilization high if segments of an array are not being processed, as for example in LU-decomposition.

Callahan and Kennedy[1] have made a different proposal for the distribution of array data. In their proposal, programmers must provide explicit mapping functions for array indices to processor numbers. In our design, these mapping functions are created automatically from much simpler allocators, while keeping the program independent of the number of physical processors. On the Connection Machine, the default allocation is equivalent to SPREAD. LOCAL or SCATTER allocations must be programmed explicitly.

## 2.6 Other extensions of Modula-2

The original definition of Modula-2 in reference [8] places several restrictions on arrays. The first concerns open arrays. An open array is an array without declared bounds. Open arrays are essential for subprograms that operate on arrays whose size is unknown until runtime. Modula-2 allows only one-dimensional, open arrays. For convenient handling of higher-dimensional arrays, open array types should be allowed to have more than one dimension. Multi-dimensional, open arrays are actually proposed for the ISO-standard of Modula-2[2].

Another troublesome restrictions involves compile-time constants. For example, the **forall** statement uses subranges, whose bounds, according to the original language definition of Modula-2, would have to be compile-time constants. This restriction is inappropriate for array parameters whose array bounds are not known until runtime. Similarly, it is often necessary to create temporary, local arrays whose size is determined by the size of

another array that is passed as a parameter. We therefore propose that constant expressions are evaluated at runtime. When a constant expression is used in a constant declaration, the expression is evaluated and used to initialize the constant. No assignments to constants are permitted. When used as an array bound, a constant expression is evaluated when the array is allocated; the array bounds remain unchanged for the lifetime of the array. Similarly, when used in a subrange of a **forall** statement, a constant expression is evaluated once and used to determine the number of processes. The constant expression is not reevaluated for the duration of the **forall** statment.

As an example, consider the procedure *Count*. *Count* returns the number of bits in a bitvector that have the value TRUE. A possible solution is to sum a vector that is initialized to 1 where the bitvector has the value TRUE, and to 0 elsewhere. This vector must be allocated at runtime, since it is unknown what size to choose at compile time.[3] It would be wasteful and awkward to require that the caller provide the array. *Count* is illustrated below.

```
PROCEDURE Count(bits: ARRAY OF BOOLEAN): CARDINAL;
VAR temp :   ARRAY[0 .. HIGH(bits)] OF INTEGER;
VAR stride:  CARDINAL;
BEGIN
   FORALL i : [0 .. HIGH(bits)] IN PARALLEL
      IF bits[i] THEN temp[i] := 1
                 ELSE temp[i] := 0
      END
   END
   (* Now compute the sum of elements of temp with *)
   (* recursive doubling, as described earlier.     *)
   ...
   RETURN temp[0];
END Count
```

Another restriction that can be lifted is that set types must have a base type with a small cardinality, for example the wordlength of a computer. With highly parallel machines, there is no rational for such a severe restriction. Instead, sets should be allowed to be as large as memory permits. Of

[3]Summing the bitvector itself would not work unless each bit occupies a follow word.

course, an implementation is free to pack a set type densely into memory words.

# 3 Implementation of the forall Statement

We consider implementing the synchronous and asynchronous forms of the **forall** statement on both synchronous and asynchronous architectures. Particular emphasis is on how to simulate a large number of processes, $p$, that potentially exceeds the number of physical processors, $P$. We assume $p > P$ in the following.

## 3.1 Process-to-processor assignment

An efficient assignment of processes to processors is important when there are thousands of processors. The assignment can be performed statically by the compiler, or dynamically by the runtime system. A static assignment has the advantage of eliminating queues of ready processes. The overhead for managing these queues might easily exceed the actual work to be done in a fine-grained parallel algorithms such as those presented earlier. On the other hand, a poor static assignment might not use the available processors well. Clearly, any reasonable process assignment must take the allocation of data to processors and the communication network into account.

As an example, consider the following program fragment.

```
A: ARRAY[1..q] SPREAD OF T;
B: ARRAY[1..r] SPREAD OF T;
FORALL i: [1..p] IN PARALLEL  A[e1(p)] := B[e2(p)]  END
```

where $e1(p)$ and $e2(p)$ are expressions in $p$. Without any further assumption about these expressions and the relations among $p$, $q$, and $r$, a reasonable assignment is to spread the $p$ processes over the same processors that are available to the larger of the arrays $A$ and $B$. Assume these are $P$ processors. Let $v = \lceil p/P \rceil$. Processor $i$ would then execute processes $vi, \ldots, v(i+1)-1$ in sequence.

Note that the process-to-processor assignment may actually change from statement to statement within a single **forall**, depending on what data structures are being accessed. Control may therefore jump processors from one statement to the next, or even within a statement. The code produced by a compiler optimizing for memory references may therefore be quite different

from the traditional method of rescheduling a process onto a potentially different processor only at synchronization events. Much research in process management on highly parallel machines remains to be done.

Many interconnection networks can treat certain communication patterns better than others. For instance, on a hypercube network, near-neighbor communication in a $n$-dimensional grid and communication over distances that are powers of 2 can be treated more efficiently than others. Suppose the index expressions in the above program fragment are linear in $p$, i.e., of the form $c_1 \times p + c_2$, where $c_1$ and $c_2$ are constants, perhaps even powers of 2. In those situations, efficient communication instructions can be chosen by a compiler optimizing for a hypercube network. Compilers for other communication networks may be able to exploit other special cases. Clearly, future research in optimizing compilers must address the problem of minimizing communication time in highly parallel machines.

## 3.2 Implementation of the asynchronous forall

The asynchronous **forall** is easy to implement. Since no assumptions about the relative speeds of the processes can be made, an implementation is free to choose any order, for example a fully asynchronous, fully synchronous, a vectorized, a sequential, or even a random order.

Recall that the asynchronous **forall** statement does not terminate until all created processes terminate. Thus, on a MIMD machine, all processes must perform a synchronization step at the end. An asynchronous reduction tree similar to the one described for summing the elements of a vector can be used to avoid linear synchronization time when processes terminate nearly simultaneously. On an SIMD machine, no such synchronization is necessary. On an MSIMD machine, only the controllers need to synchronize.

An important issue on an SIMD machine is how to fully use the available processors. Recall that whenever control flow splits into two or more branches, only one set can be fed instructions at a time, while the other sets idle.

To avoid the idling of processors, more sophisticated scheduling algorithms are possible. The goal is to assign processes to processors in such a fashion that nearly all processors remain busy. This goal can be accomplished with a simple rescheduling at every branchpoint. For example, consider an **if** statement with two arms, and assume that each of the $P$ physical processors is assigned $v$ processes. The usual simulation is to feed both arms of the **if** statement $v$ times to the processors, effectively idling half of the

processors. Instead, consider the following scheme. First, each processor evaluates the condition for all its assigned processes and divides them into two sets, depending on the results. Next, all $P$ processors select processes with value TRUE an then receive the instructions for the corresponding arm from the controller. When all processors are done with processes containing the value TRUE, the controller switches to the other arm. For sufficiently high $v$ and evenly distributed values of the conditions, few processors will actually idle, achieving nearly full utilization. Note that such a scheduling is valid because the asynchronous **forall** makes no assumptions about the relative order of the two arms of the **if** statement.

On an MSIMD machine, several branches can be executing in an overlapped fashion, until the number of parallel branches exceeds the number of available controllers. Up to that point, all processors can be kept busy. After that, the remaining branches are executed in SIMD fashion, possibly with rescheduling as discussed above.

Finally, the asynchronous **forall** can be executed efficiently on vector computers, since order is immaterial. For simulation on sequential machines, all that needs to be done is to replace the **forall** statement with a **for** statement over the same range. Note, however, that such a simple-minded simulation may mask many potential programming errors. Perhaps a randomized order of the processes is more appropriate in helping programmers with detecting errors when testing parallel programs on sequential machines.

### 3.3 Implementation of the synchronous forall

Clearly, a MSIMD machine can implement the synchronous **forall** statement directly. When the number of parallel branches exceeds the number of controllers, the remaining branches must be simulated by multiplexing the available processors, as is done on SIMD machines. Since separate control flow branches execute asynchronously, efficient process scheduling is possible, as explained in the previous section.

Achieving synchronous execution on an MIMD machine can be expensive. A simple, but inefficient simulation would be to insert a synchronization command after every instruction. Then each instruction would essentially take time proportional to the logarithm of the available processors, which would slow down all programs significantly. Fortunately, a synchronization command is not needed for every instruction, but only before and after every memory write. This synchronization suffices because processes are affected by other processes only through changes in memory. Neverthe-

less, a modest amount of hardware for simulating SIMD, such as a global clock for instruction execution, would provide a much more efficient implementation of synchronous execution.

If the number of processes exceeds the number of processors, it is important that the multiplexing does not violate the semantics of the synchronous **forall**. Consider the following statement.

```
FORALL i:[....] IN SYNC x[i+1] := x[i] END
```

If process $i$ actually executes before process $i+1$, a naive implementation would produce incorrect results, even on a SIMD machine. Instead, all processes have to follow faithfully the steps in the definition of the synchronous execution of the assignment statement. Each process must first evaluate the left and right sides of the assignment before any write to memory takes place. This means that each process must save both a pointer and a value and wait for all processes to complete their evaluation of the left and right hand sides before the actual assignment. Since evaluating the right and left sides might cause side effects (through function calls, for instance), these computations must be carried out such that they appear synchronously even if there are more processes than processors.

Correct, synchronous execution requires overhead in space and time. This overhead can be reduced on both SIMD and MIMD architectures if the synchronous **forall** can be transformed into the asynchronous form with no or infrequent synchronization.

# 4 Conclusions

We have presented simple language constructs for writing highly parallel, machine-independent programs. These constructs can be implemented efficiently on SIMD, MSIMD, and MIMD computers. Simple, machine-independent control over the mapping of data to processors allows compilers to optimize communication time on architectures with distributed memory. Work on optimizing compilers for a highly parallel machine is in progress.

# References

[1] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[2] BSI Modula-2 Standardisation Working Group. First working draft Modula-2 standard. 1989.

[3] W. Daniel Hillis. *The Connection Machine.* The MIT Press, 1985.

[4] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12), Dec. 1986.

[5] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL Language Reference Manual.* Lawrence Livermore National Laboratory, March 1985.

[6] James R. McGraw. The val language: description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[7] Horst D. Simon, editor. *Scientific Applications of the Connection Machine.* World Scientific Publishing Co., 1989.

[8] Nilaus Wirth. *Programming in Modula-2.* Springer Verlag, third, corrected edition, 1985.